The Pre-Execution Interpretation Framework
By Irina Akopova
Full-Spectrum Salesforce Engineer

Assumption-Surfacing:
A Pre-Execution Validation Layer for Agentic Systems
(Prompt pseudo-complier and testing modes)

♦ The Shift (The Interface)

Instead of software engineers writing exact, line-by-line code, enterprise platforms (like **Salesforce Headless 360**) are moving toward a model where users describe their goals in plain language or structured business terms. The AI then instantly translates this intent, building workflows, configurations, and operational logic to make it happen.

This evolution introduces a fundamentally new system pipeline:

Natural Language → Probabilistic Interpretation → Deterministic Execution

While this model drastically increases development speed and abstraction, it also introduces a critical, unmonitored architectural layer. Because the transition relies on a probabilistic model, the interpretation stage effectively becomes a hidden design surface where final system behavior is shaped out of sight.

♦ The Hidden Problem (The Evolution of Ambiguity)

In modern Salesforce-style architectures, system behavior is increasingly generated from human-readable language, but this is not unconstrained "natural language." Instead, it functions as a compressed, instruction-layer programming interface that evolves toward structural formality to reduce ambiguity.

We can see this evolution by dragging a single task through three levels of specification:

- **Pure Natural Language:** *"Hey, we need to make sure we don't lose customers, so escalate alerts if engagement drops."*
  This is highly conversational and completely unconstrained.

- **Instruction Language:** *"Create a customer renewal alert and escalation flow based on contract status and engagement signals."*
  This is more formal, resembling a compressed prompt requirement.
- **Programmatic Specification (DSL):** *if (Contract.Status == 'At Risk' && Customer__c.EngagementSignal__c < threshold) { triggerEscalation(); }*
  This is executable programming logic expressed in a DSL-style syntax.

As a developer moves from conversational text to strict, programmatic instructions, the interface becomes more efficient. However, increased structural formality does not actually eliminate ambiguity; it merely relocates it upward into hidden interpretation boundaries and unresolved system constraints.

Even in the highly structured, code-like format, the system must still infer missing operational realities. For instance, what exactly constitutes the *EngagementSignal__c* data payload, and how reliable is its source? What numerical value fills the threshold variable? If the *triggerEscalation()* function runs, who takes organizational ownership of the resulting task if a field is missing, and what are the timing thresholds for SLA breaches?

When the final system is generated, it often looks correct and runs without errors, which hides the fact that the AI had to infer missing rules during interpretation. The issue is not that human-readable input is inherently unclear. The issue is that more structured formats can create the illusion that ambiguity has been removed, when in reality it has just moved into hidden interpretation steps.

♦ The Shift in Focus (The Fix)

Traditional QA focuses on whether the final system runs correctly. In agentic systems, that is not enough. A workflow can execute without errors while still being built on missing or incorrect assumptions introduced during interpretation.
The failure point is not execution, but translation: how intent becomes system behavior.

To address this, assumption-surfacing can be treated as a pre-execution layer that behaves like a pseudo-compiler for intent. Just as a compiler checks code before build, this layer forces the model to surface assumptions, missing context, and uncertainty boundaries before any workflow is generated.

Instead of silently resolving ambiguity, the system exposes it as a set of “interpretation warnings” prior to execution. This can be implemented as a meta-instruction inside the prompt that requires the model to inspect its own translation process before producing artifacts.

For example, a developer may append a validation directive to the system instruction layer:

```
[PSEUDO_COMPILER_MODE = ACTIVE]

Before generating Salesforce metadata, Apex logic, Flow structures, or automation artifacts:
```

1. Identify implicit assumptions required for the requested system behavior to function correctly.
2. Surface unresolved business rules or undefined operational constraints.
3. Identify missing schema dependencies or contextual variables that materially affect execution outcomes.
4. Output findings as a structured "Interpretation Warnings" report before generation proceeds.
5. Pause for human validation.

When the developer submits the original request: ***"Create a customer renewal alert and escalation flow based on contract status and engagement signals."***

The system emits a pre-execution validation layer instead of proceeding directly to generation:

>>> INTERPRETATION WARNINGS

[1] Undefined Business Threshold
The term "engagement signals" is not operationally defined.
Current interpretation assumes:
- 20% decrease in login activity over 14 days
- No weighting applied to support activity or email engagement

This assumption materially affects escalation triggering behavior.

[2] Missing Ownership Resolution Logic
Escalation routing behavior is undefined if:
- Account Owner is inactive
- Customer Success Manager field is null
- Multiple teams share account responsibility

Default routing behavior has not been specified.

[3] Missing Schema Dependency
Historical contract status tracking was assumed to exist.
If status history is unavailable, renewal trend analysis may produce incomplete or inaccurate escalation timing.

[4] Undefined Notification Constraints
Delivery assumptions were inferred for:
- Slack notifications
- Email escalation timing
- After-hours handling

No communication policy or SLA boundaries were provided.

>>> GENERATION PAUSED
Confirm assumptions or provide additional constraints before workflow synthesis proceeds.

By instructing the LLM to act as its own compiler, the system gives the admin/developer a chance to see the exact conditions under which the generated output remains valid. It transforms a conversational black box into an inspectable, deterministic engineering checkpoint.
In practice, assumption-surfacing does not guarantee discovery of all hidden dependencies. A model may fail to surface assumptions it also failed to recognize during synthesis.

For higher-assurance workflows, assumption evaluation may require separation between the generation layer and the evaluation layer, including independent evaluator prompts, retrieval grounding, or specialized validation agents operating with different objectives.

♦ Validation Orientations (Application Modes)

Once assumption-surfacing is introduced as a pseudo-compiler layer, it can be used in two primary contexts: system synthesis and behavior testing.
The underlying mechanism does not change. What changes is *what is being validated* - generated system architecture or model behavior under interpretation.

In both cases, the key shift is the same: validation moves from post-generation execution checks to pre-generation interpretation checks.

○ Headless / Build-Time Compilation Mode (Prompt pseudo-complier)

In Headless-style environments such as **Salesforce Headless 360**, the pseudo-compiler is applied before workflows, metadata, or automation are generated from high-level intent.

Instead of immediately producing system artifacts, the model first exposes interpretation assumptions, missing constraints, and ambiguous business rules that would otherwise be silently embedded in the generated system.

Example:

Developer instruction:	*"**Create a customer retention workflow for disengaged users.**"*
Compiler warnings:	• *"Disengaged" is undefined. I assumed login inactivity over 30 days.* • *Retention priority logic is missing for enterprise vs standard customers.* • *Customer communication channel preference is not specified.*

○ Behavior Pre-Execution Mode (Prompt testing)

In standard **prompt engineering or Agentforce**-style testing, the same pseudo-compiler is used not to build systems, but to evaluate how robust the model's interpretation is under missing or ambiguous context.
The goal here is not deployment safety, but interpretation transparency **-** understanding how the model fills gaps before its response is trusted.

Example:

Administrator instruction to bot / assistant:	***"Suggest resort activities for today based on guest interest fields: Contact.Interest1__c, Contact.Interest2__c, Contact.Interest3__c."***
Compiler output:	• *Guest age is assumed to be within a range suitable for all listed activities unless explicitly provided.* • *Physical ability is assumed to be unrestricted unless a constraint is explicitly stored in the profile.* • *Social engagement preference (group vs solo participation) is assumed neutral in the absence of explicit data.* • *Availability is assumed for all listed time slots, regardless of competing commitments.* • *Accessibility, dietary, or health constraints are assumed absent unless explicitly modeled in the data schema.*

The system does not infer "better answers." It silently resolves missing structure by inserting default assumptions that are not visible in either the prompt or the data model.
These surfaced assumptions can also inform prompt and schema evolution - for example, administrators may decide to extend the input model (e.g., adding fields such as *Contact.DOB*, *Contact.FoodAlergies__c,* accessibility indicators, or social preference attributes) so that future outputs rely less on implicit inference and more on explicit data.
Conceptually, this resembles traditional software engineering validation patterns, except applied to probabilistic synthesis instead of deterministic code construction:

Traditional Systems	Agentic Systems
Compiler warnings	Assumption warnings
Static analysis	Interpretation analysis
Type validation	Context validation
Build-time checks	Pre-execution synthesis checks

♦ Assumption Categories (Examples)

In practice, interpretation-stage assumptions tend to cluster into recurring operational categories. Examples include:

- **Schema assumptions** - assumed existence, quality, or meaning of fields and relationships
- **Threshold assumptions** - inferred numerical limits, scoring boundaries, or trigger conditions
- **Ownership assumptions** - inferred responsibility, escalation paths, or workflow routing
- **Temporal assumptions** - inferred timing windows, expiration periods, or SLA expectations
- **Policy assumptions** - inferred compliance, legal, or organizational governance rules
- **Safety assumptions** - inferred physical, financial, or operational safety conditions

- **Context assumptions** - inferred user intent, preferences, capabilities, or environmental conditions

While assumption categories vary by domain, recurring patterns suggest that interpretation-stage dependencies may be systematically classifiable rather than purely ad hoc.

♦ Materiality Threshold and Human Control Boundary (Human-in-a-Loop)

A key requirement of assumption-surfacing is that not all uncertainties are equally relevant. Without constraints, the system risks over-generating low-value or pedantic warnings, shifting from useful governance to interpretive noise.
To prevent this, assumption-surfacing operates under a **materiality threshold**: only assumptions that materially affect execution outcomes, security, compliance, or operational correctness are surfaced. This creates a practical filtering boundary between informational ambiguity and actionable risk.

In addition, final interpretation of relevance cannot be fully delegated to the model itself. A human operator (developer, admin, or architect) remains in the final arbitration layer that determines whether surfaced assumptions are meaningful, ignorable, or require schema or logic changes.
This preserves the system's intent as a **decision-support and governance layer**, not an autonomous validation authority.

♦ The Value (Observability)

Traditional governance mechanisms validate deterministic artifacts after they are generated, using processes such as code reviews, configuration checks, and security scans. In agentic synthesis systems, however, critical design flaws are often introduced earlier—during the interpretation phase—before any final artifacts exist. This shifts part of enterprise governance upstream.
Assumption-surfacing does not replace deterministic engineering practices, schema design, or formal validation. It exposes hidden interpretation dependencies before those controls are applied.
This shifts the human-AI design model from implicit interpretation and direct execution to a more explicit pipeline:

Intent Capture → Surfaced Assumptions → Human Validation → Deterministic Execution

The result is not just better output control, but a more inspectable system design process, where administrators and developers retain visibility into how AI-generated workflows were derived and what assumptions they depend on.